Journal of Universal Computer Science, vol. 6, no. 12 (2000), 1203-1225
submitted: 28/2/00, accepted: 13/10/00, appeared: 28/12/00

Some Elements of Z Specification Style: Structuring Techniques

Anthony MacDonald

Department of Computer Science and Electrical Engineering
and
Software Verification Research Centre
The University of Queensland
Brisbane 4072, Australia
email: {anti}@csee.uq.edu.au

David Carrington

Department of Computer Science and Electrical Engineering
and
Software Verification Research Centre
The University of Queensland
Brisbane 4072, Australia
email: {davec}@csee.uq.edu.au

Abstract: This article investigates the issue of structuring Z specifications. It uses examples from a large specification (the production cell) to examine both conventions for using Z and notational extensions, including Object-Z. Because of the importance of good structure within a specification, specifiers need to be aware of a range of structuring techniques and understand where each is applicable.

Key Words: Formal specification, Z notation, specification structure.

Category: D.2.1, F.3.1, F.4.3.

1 Introduction

[Kernighan and Plauger, 1978] assert that "Good programming cannot be taught by teaching generalities. The way to learn to program well is by seeing, over and over, how real programs can be improved by the application of a few principles of good practice and a little common sense". We feel this applies equally to specifications, and in this article we attempt to highlight the suitability of differing structuring techniques for particular situations.

Application of formal methods and, in particular, Z [Toyn, 1999, Spivey, 1992] is increasing. Z is a notation rather than a specific method and the notation provides many ways to specify systems. This flexibility raises a question: is the structure of the specification important? To answer this question, we consider what Z specifications aim to achieve and whether satisfying these aims is assisted by appropriate choices of structure.

[Spivey, 1989] states that formal specifications provide a precise description of a system without unduly constraining how the system achieves it. Specifications allow questions about the system to be answered without having to "disentangle the information from a mass of detailed program code, or to speculate about the meaning of phrases in an imprecisely worded prose description". Spivey also states that a formal specification can act as the single reference point between customer, designer and programmer. Specifications are supposed to capture system requirements clearly and concisely. Formal specifications aim to provide a concise and unambiguous description of a problem and should allow clear communication of the expected system behaviour.

Can specification structure help achieve these aims? Normal Z practice is to structure a specification as formal schemas embedded in informal explanatory text. However with Z's increasing popularity, it is being used in a larger range of situations and the size of specifications is increasing. The problems faced when writing/reading large specifications often mirror those associated with large pieces of programming code. Just as techniques were developed and used to structure code at a higher level than procedures, techniques must be used to structure large specifications at a higher level than schemas. Approaches applicable to large software systems may be applicable to large specifications. An *'in the large'* structuring technique should enable large specifications to satisfy the aims of formal specifications. Any additional structuring technique should complement using schemas for lower-level structure. The components of the specification should be independently understandable and should compose easily to form an understandable whole without reference to unnecessary detail. Unfortunately there is no single structuring technique: different techniques suit different situations. This article demonstrates six different structuring techniques to highlight the choices available to specifiers, starting with the simplest, or flat style. A logical progression from this is the "Oxford" style which partitions the specification into components and which can be seen in [Hayes, 1993, Chapter 2]. The third style parameterises similar components to avoid duplication. [Hayes, 1993, Chapter 5] provides an example. Two further techniques use extensions to Z to impose structure: the first uses a library/module extension of Z developed by [Hayes, 1993, Hayes and Wildman, 1993] and the second uses Object-Z [Duke and Rose, 2000, Smith, 1999, Rose, 1992]. Combining object-orientation with Z has been investigated by many people, including [Schuman and Pitt, 1987] and [Hall, 1990]. Two collections [Lano and Houghton, 1994, Stepney *et al.*, 1992] describe work on extending Z with object-oriented features. These five specification styles can be categorised as bottom-up specification styles and focus on building large specifications from smaller specification components. A final style of specification is also introduced that builds the specification top-down. This final style of specification is similar to the flat style, but provides another view. We believe each of the six specifications are equivalent in the behaviour they define, but this has not been formally verified.

The article is divided into the following sections. [Section 2] introduces the production cell, which is used as the basis of our study. [Section 3] introduces the bottom-up specification structuring techniques using examples from a specification of the production cell case study, and [Section 4] introduces the top-down specification technique. [Section 5] discusses the suitability of each technique when applied to both small and large specifications. The article closes with conclusions from the discussion.

2 Production cell

The original production cell can be found in a metal processing plant in Karlsruhe, Germany. Forschungszentrum Informatik (FZI), Karlsruhe, used the production cell as the basis for a study [Lewerentz and Lindner, 1995] of formal methods for critical software systems. The first step of that study generated a requirements document [Lewerentz and Lindner, 1995, Chapter 2] that attempts to capture, in plain language, the specification of a software controller for the production cell. The software controller is required to control a simplified simulation [see Fig. 1] of the production cell which runs cyclically.

The production cell simulation (the system) takes a metal blank as input. The metal blank is transported by a conveyer belt (the feed belt) to an elevating rotary table. The elevating rotary table rotates and rises vertically to present the metal blank for the robot to pick up. The robot takes the metal blank to a press. The press presses the metal blank and the robot retrieves the metal blank. The robot transports the metal blank to a second conveyer belt (the deposit belt). The deposit belt transports the metal blank to the crane.

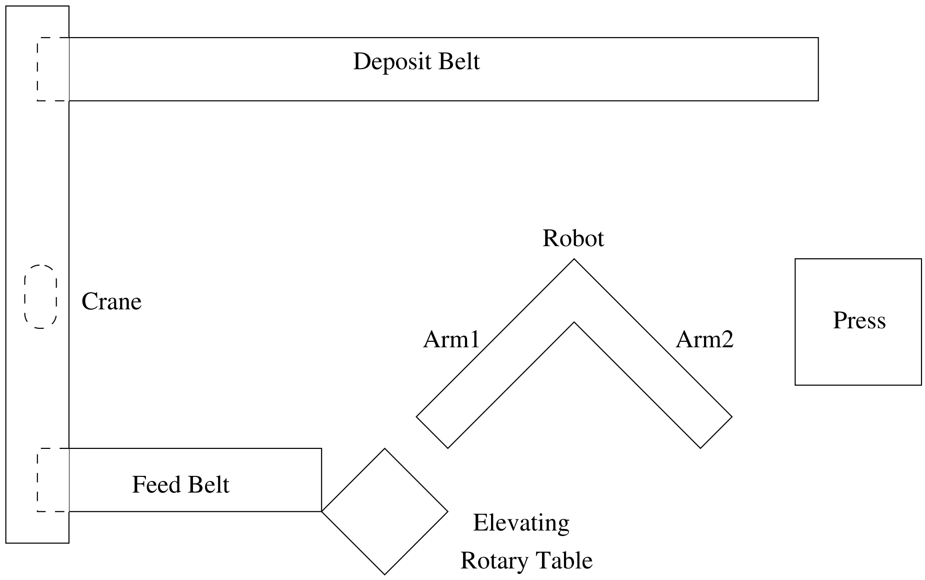

Figure 1: The production cell

The crane picks up the metal blank and transports the metal blank to the feed belt and completes the cycle.

The system actions are further complicated by a desire for speed and efficiency. To accommodate these aims, the robot is fitted with two arms. The arms are placed at right angles to each other and rotate together. The arms can extend/retract and load/unload independently. A second consequence of the aim for speed and efficiency is that the system should be able to handle multiple metal blanks in different parts of the system concurrently.

Safety and liveness requirements also constrain the system's operation. Safety requirements are most important in this setting, as violation of them can lead to human injury or machine damage. Violation of liveness results in safe failure in the worst case. The safety requirements include restricting machine mobility to within safe limits, avoiding collisions between machines, not dropping metal blanks outside safe areas and keeping metal blanks sufficiently distant.

The specifications all make the following assumptions:

– all component movement is discrete, and
– conveyor belt movement is hidden and the belts are restricted to holding a single metal blank each.

3 Bottom-up Z specification styles

Our Z specification [MacDonald and Carrington, 1994][1995][MacDonald, 1998][1] of the system has been built in a bottom-up manner. The first part of the specification is the independent specification of each component. The system is subdivided into the

[1] [MacDonald and Carrington, 1994] is a complete production cell specification; this article is extracted from [MacDonald and Carrington, 1995] and [MacDonald, 1998].

following components: feed belt, elevating rotary table, robot, press, deposit belt and crane. The independent specification of each component captures the local state information and the allowable operations on the state. The independent specifications do not take into account relationships between different components. For example, the robot cannot pick up a metal blank from the elevating rotary table if no metal blank is there; however this information is not relevant to the independent specifications of either the robot or the elevating rotary table. The independent specifications, each with their own local state, are combined to give the overall state of the system. Operations from the independent specifications are promoted to apply to the total state and are adjusted to take into consideration the relationships between components. The system is modelled by the overall state and the corresponding operations, but the operations are partial and are only guaranteed to succeed (i.e., have a specified behaviour) when their pre-conditions are satisfied. The final section of the specification extends the partial operations to total operations. This is achieved by specifying possible error cases for each partial operation as an error schema. Joining the partial operation and the error schema via the schema calculus provides the final operation.

The five structuring techniques are explained using the robot section of the specification. The robot has several components and is introduced using the flat style. The other techniques are introduced relative to this initial specification.

3.1 Flat

The robot has two arms, called arm1 and arm2, and four discrete orientations.

1 - *load_arm*1 is where arm1 is aligned with the elevating rotary table and arm2 is not aligned with any other production cell component,
2 - *load_arm*2 is where arm2 is aligned with the press and arm1 is not aligned,
3 - *unload_arm*2 is where arm2 is aligned over the deposit belt for unloading and arm1 is not aligned, and
4 - *unload_arm*1 is where arm1 is aligned with the press and arm2 is not aligned.

$$Robot\_Orientation ::= load\_arm1 \mid load\_arm2 \mid unload\_arm2 \mid unload\_arm1$$

Each robot arm has two attributes: the first, *Component_Loaded*, records whether the robot's arm is carrying a blank (*loaded*) or not (*unloaded*) and the second, *Arm_Extent*, records whether the arm is *retracted* or *extended*.

$$Component\_Loaded ::= loaded \mid unloaded$$

$$Arm\_Extent ::= retracted \mid extended$$

Initially, both arm1 and arm2 are *unloaded* and *retracted* and the robot's orientation is *load_arm*1.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Robot\;}} \\
robot_orientation : \\
\qquad\qquad Robot_Orientation \\
arm1_store : Component_Loaded \\
arm2_store : Component_Loaded \\
arm1_extent : Arm_Extent \\
arm2_extent : Arm_Extent \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Init_Robot\;}} \\
Robot \\
\hline
robot_orientation = load_arm1 \\
arm1_store = unloaded \\
arm2_store = unloaded \\
arm1_extent = retracted \\
arm2_extent = retracted \\
\hline
\end{array}$$

For safety reasons, the robot must not rotate if either arm is extended. The robot's rotation does not change either the arm store status or the arm extension status, but only accesses the relevant variables.

$$
\begin{array}{|l}
\hline
\textit{Rotate\_Robot\_0} \\
\Delta Robot \\
new\_pos? : Robot\_Orientation \\
\hline
arm1\_extent = retracted \land arm2\_extent = retracted \\
robot\_orientation' = new\_pos? \\
arm1\_store' = arm1\_store \land arm2\_store' = arm2\_store \\
arm1\_extent' = arm1\_extent \land arm2\_extent' = arm2\_extent \\
\hline
\end{array}
$$

Arm1 operations do not change the robot's orientation or attributes of arm2.

$$
\begin{array}{|l}
\hline
\Delta Arm1 \\
\Delta Robot \\
\hline
robot\_orientation' = robot\_orientation \\
arm2\_store' = arm2\_store \\
arm2\_extent' = arm2\_extent \\
\hline
\end{array}
$$

To extend arm1, the robot must be at one of two orientations: *load_arm*1 or *unload_arm*1. At this level of abstraction, interactions between the arm and either the elevating rotary table or the press are ignored. The store state of the arm is not changed. The result of an *Extend_Arm*1_0 operation is the arm extended. A successful retract operation leaves arm1 retracted without changing the store state of the arm or moving the robot.

$$
\begin{array}{|l}
\hline
\textit{Extend\_Arm1\_0} \\
\Delta Arm1 \\
\hline
robot\_orientation \in \\
\quad \{load\_arm1, unload\_arm1\} \\
arm1\_store' = arm1\_store \\
arm1\_extent' = extended \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Retract\_Arm1\_0} \\
\Delta Arm1 \\
\hline
arm1\_store' = arm1\_store \\
arm1\_extent' = retracted \\
\hline
\end{array}
$$

Loading arm1 only occurs when the robot is at orientation *load_arm*1 and the arm is extended and unloaded. A load arm1 operation causes the arm to be loaded with no other change. Unloading arm1 has as its pre-condition that the orientation is *unload_arm*1 and the arm loaded, and as its post-condition that the arm is unloaded.

$$
\begin{array}{|l}
\hline
\textit{Load\_Arm1\_0} \\
\Delta Arm1 \\
\hline
robot\_orientation = load\_arm1 \\
arm1\_store = unloaded \\
arm1\_extent = extended \\
arm1\_store' = loaded \\
arm1\_extent' = arm1\_extent \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Unload\_Arm1\_0} \\
\Delta Arm1 \\
\hline
robot\_orientation = unload\_arm1 \\
arm1\_store = loaded \\
arm1\_extent = extended \\
arm1\_store' = unloaded \\
arm1\_extent' = arm1\_extent \\
\hline
\end{array}
$$

Operations for arm2 are almost identical except that they occur at orientations *load_arm*2 and *unload_arm*2 instead of orientations *load_arm*1 and *unload_arm*1. They are not shown since they do not contribute directly to the discussion. However, the existence of multiple arms is important to the discussion.

3.2 Partitioned

The second technique partitions the robot into components. Each arm is specified and then included in the robot[2].

$$\begin{array}{|l}
\textit{Arm1} \\
\hline
arm1_store : Component_Loaded \\
arm1_extent : Arm_Extent \\
\hline
\end{array}$$

The given type *Robot_Orientation* must be declared before the arms as the arm operations, except for retraction, are orientation-dependent. As access to the robot attribute *robot_orientation* is not possible, the operations require an input parameter. The input parameter, *current_pos?*, contains the arm's current orientation and is used to ensure that extending, loading and unloading of an arm occurs at the correct orientations.

$$\begin{array}{|l}
\textit{Extend_Arm1} \\
\hline
\Delta Arm1 \\
current_pos? : Robot_Orientation \\
\hline
current_pos? \in \\
\quad \{load_arm1, unload_arm1\} \\
arm1_store' = arm1_store \\
arm1_extent' = extended \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Load_Arm1} \\
\hline
\Delta Arm1 \\
current_pos? : Robot_Orientation \\
\hline
current_pos? = load_arm1 \\
arm1_store = unloaded \\
arm1_extent = extended \\
arm1_store' = loaded \\
arm1_extent' = arm1_extent \\
\hline
\end{array}$$

Arm2 is defined similarly and is omitted for brevity.

The robot's state schema is composed of three component state schemas. Defining the schema *Robot_0* is unnecessary here but simplifies later promotion operations. An extra operation, *Generate_Orientation*, outputs the robot's current orientation for input to the arm operations.

$$Robot\_0 \mathrel{\widehat{=}} [robot\_orientation : Robot\_Orientation]$$

$$\begin{array}{|l}
\textit{Robot} \\
\hline
Robot_0 \\
Arm1 \\
Arm2 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Generate_Orientation} \\
\hline
\Xi Robot \\
current_pos! : Robot_Orientation \\
\hline
current_pos! = robot_orientation \\
\hline
\end{array}$$

Promoting operations on the arms to the robot state is straight-forward and involves a simple framing schema and the schema calculus. The framing schema for arm1 (*Arm1_Ops*) constrains the state variables that an arm1 operation can change to only those from the *Arm1* state.

$$Arm1\_Ops \mathrel{\widehat{=}} \Delta Robot \land \Xi Robot\_0 \land \Xi Arm2$$

$$Load\_Arm1\_0 \mathrel{\widehat{=}} Generate\_Orientation \gg (Arm1\_Ops \land Load\_Arm1)$$

Without the input parameter to the arm1 operations, separate framing schemas are required for each operation, making the specification larger and more complicated. The robot's arms are identical in function and differ only in the orientations that an arm can be extended, loaded or unloaded. The next three structuring styles specify a generic arm which is instantiated twice.

[2] The complete specification of the arms and the robot are found in the appendix of [MacDonald, 1998].

3.3 Parameterised

The parameterised specification defines a generic arm which is instantiated twice in the robot[3]. A generic arm is specified as an independent entity parameterised on *Orientation*. An arm does not contain information to restrict its actions to specific orientations but captures the existence of such orientations. The arm state schema contains two new state variables, *load_pos* and *unload_pos* which restrict the orientations at which loading and unloading of an arm can occur. The variables *load_pos* and *unload_pos* are never changed (after initialisation) and *ΔArm* is redefined to equate these variables (replacing the conventional definition).

$$\begin{array}{|l}
\hline
\textit{Arm}[\textit{Orientation}] \\
\textit{store} : \textit{Component_Loaded} \\
\textit{extent} : \textit{Arm_Extent} \\
\textit{load_pos} : \mathbb{P}\,\textit{Orientation} \\
\textit{unload_pos} : \mathbb{P}\,\textit{Orientation} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\Delta\textit{Arm}[\textit{Orientation}] \\
\textit{Arm}[\textit{Orientation}] \\
\textit{Arm}'[\textit{Orientation}] \\
\hline
\textit{load_pos}' = \textit{load_pos} \\
\textit{unload_pos}' = \textit{unload_pos} \\
\hline
\end{array}$$

The arm operations differ slightly from those specified in the partitioned version. Apart from the obvious addition of a generic parameter, an extend operation can always occur at this level of abstraction, i.e. *Extend* has no pre-conditions and the post-condition is an extended arm. The load and unload operations compare the input *current_pos*? with the variable *load_pos*/*unload_pos* to ensure the operations proceed at valid orientations. The specifications of *Extend* and *Load* highlight two possible methods of defining operations within a component. The specification of *Extend* defers decisions on when an extension is suitable to the caller of the operation. The specification of *Load* however defines that loading should always be constrained to specific orientations and that the caller will define these orientations. Both *Extend* and *Load* could have been specified in either way, but they are specified differently to highlight different methods within the parameterised style.

$$\begin{array}{|l}
\hline
\textit{Extend}[\textit{Orientation}] \\
\Delta\textit{Arm}[\textit{Orientation}] \\
\hline
\textit{extent}' = \textit{extended} \\
\textit{store}' = \textit{store} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Load}[\textit{Orientation}] \\
\Delta\textit{Arm}[\textit{Orientation}] \\
\textit{current_pos}? : \textit{Orientation} \\
\hline
\textit{current_pos}? \in \textit{load_pos} \\
\textit{extent} = \textit{extended} \\
\textit{store} = \textit{unloaded} \\
\textit{extent}' = \textit{extent} \\
\textit{store}' = \textit{loaded} \\
\hline
\end{array}$$

Compared to the flat specification, the robot's state schema changes in two ways: the number of state variables decreases by hiding arm details within the generic arm schema and predicates constraining where each arm can be extended and loaded/unloaded are added. This is an example of a common trade-off to be made when writing Z specifications; either putting constraints as part of a state invariant or putting them on every operation that might otherwise violate them. Extra constraints on the arms are needed because the arm specification is independent of the robot. For example, at the level of an arm it is logical to allow extension unconditionally, but for safety reasons the robot allows arm extension at certain orientations only.

[3] The complete specification of the arm and the robot can be found in [MacDonald and Carrington, 1994].

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Robot}} \\ \hline
robot_orientation : Robot_Orientation \\
arm1, arm2 : Arm[Robot_Orientation] \\ \hline
arm1.extent = extended \Leftrightarrow robot_orientation \in \{load_arm1, unload_arm1\} \\
arm1.load_pos = \{load_arm1\} \wedge arm1.unload_pos = \{unload_arm1\} \\
arm2.extent = extended \Leftrightarrow robot_orientation \in \{load_arm2, unload_arm2\} \\
arm2.load_pos = \{load_arm2\} \wedge arm2.unload_pos = \{unload_arm2\} \\ \hline
\end{array}$$

The robot's operations change in several ways. Framing schemas[4] are used to promote the arm operations to the robot state. These restrict an operation to apply to either *arm*1 or *arm*2. The framing schema for arm1, *Arm*1*_Ops*, equates both *robot_orientation* and *arm*2 and binds *arm*1 and *arm*1′ to *Arm* and *Arm*′ respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Arm1_Ops}} \\ \hline
\Delta Robot \\
\Delta Arm[Robot_Orientation] \\ \hline
arm2' = arm2 \\
robot_orientation' = robot_orientation \\
arm1 = \theta Arm \\
arm1' = \theta Arm' \\ \hline
\end{array}$$

Binding *arm*1 to *Arm* is necessary as the expression *arm*1.*extent* is valid, while *arm*1.*Load* is not. By including both $\Delta Robot$ and $\Delta Arm[Robot\_Orientation]$ and binding *arm*1 to *Arm*, the framing schema allows the *Load* operation on an *Arm* to be promoted and used at the robot level on *arm*1.

$$\begin{array}{l}
Load_Arm1_0 \ \widehat{=} \ Generate_Orientation \gg \\
\qquad ((Arm1_Ops \wedge Load[Robot_Orientation]) \setminus (\Delta Arm[Robot_Orientation]))
\end{array}$$

Using the *Generate_Orientation* schema defined in [Section 3.2], the current orientation is piped to the load arm operation. The arm operation is restricted to changing only *arm*1's state variables with all $\Delta Arm[Robot\_Orientation]$ occurrences hidden ($\Delta Arm[Robot\_Orientation]$ is present in both *Arm*1*_Ops* and *Load*[*Robot_Orientation*]).

The preceding techniques use standard Z. The next two techniques use extensions to Z to structure the specification.

3.4 Library

Specifying the robot using libraries is a logical extension of the parameterised specification. An arm library [see Fig. 2] is specified and used by the robot[5].

The library is parameterised by the type *Orientation*. Parameterisation is applied to the library as a whole rather than the individual schemas and simplifies the use of genericity. The library is also textually enclosing allowing the *load_pos* and *unload_pos* state variables of the parameterised version to be constants within the library. A special definition of ΔArm is not required. The major differences between the library style and the parameterised style occur at the robot level when the arm is used. Prior to use, a library must be instantiated. As the specification requires two arms, the instantiated

[4] This technique requires an additional framing schema to be used at initialisation. See [MacDonald and Carrington, 1994].

[5] The complete specification of the robot is in the appendix of [MacDonald, 1998].

library *Arm*[*Orientation*]

An arm has two state variables. The first, *store*, records whether an arm is loaded or unloaded and an arm is initially unloaded. The second, *extent*, records whether the arm is retracted or extended and is initialised to retracted. Associated with an arm are two constants, *load_pos* and *unload_pos*, which are the orientations where an arm can be loaded or unloaded. The values of the constants are set at instantiation time.

$Arm\_Extent ::= retracted \mid extended$

$load\_pos, unload\_pos : \mathbb{P}\ Orientation$

State

$store : Component\_Loaded$
$extent : Arm\_Extent$

Init

$State$

$store = unloaded$
$extent = retracted$

Arm extension and retraction always succeed (at this level of abstraction) and change the value of *extent* while leaving *store* unchanged.

Extend

$\Delta State$

$extent' = extended$
$store' = store$

Retract

$\Delta State$

$extent' = retracted$
$store' = store$

An arm can only load and unload at certain orientations and to perform a *Load* or *Unload* operation, the arm must compare the current arm orientation to the set of allowable load or unload orientations. These operations change only the *store* variable of the arm.

Load

$\Delta State$
$current\_pos? : Orientation$

$current\_pos? \in load\_pos$
$extent = extended$
$store = unloaded$
$extent' = extent$
$store' = loaded$

Unload

$\Delta State$
$current\_pos? : Orientation$

$current\_pos? \in unload\_pos$
$extent = extended$
$store = loaded$
$extent' = extent$
$store' = unloaded$

endlib

Figure 2: The arm library

library must be decorated. The values of *load_pos* and *unload_pos* can be set when the arm is instantiated.

instantiate $Arm1 :: Arm[Robot\_Orientation]$
$Arm1 :: load\_pos = \{load\_arm1\}$
$Arm1 :: unload\_pos = \{unload\_arm1\}$

The robot's state schema consists of three component state schemas.

$Robot\_0 \mathrel{\widehat=} [robot\_orientation : Robot\_Orientation]$

Robot

$Robot\_0$
$arm1\text{-}Arm1 :: State$
$arm2\text{-}Arm2 :: State$

$arm1\text{-}extent = extended \Leftrightarrow robot\_orientation \in \{load\_arm1, unload\_arm1\}$
$arm2\text{-}extent = extended \Leftrightarrow robot\_orientation \in \{load\_arm2, unload\_arm2\}$

The qualification $arm1$- ensures all variables of $Arm1 :: State$ are prefixed with $arm1$-. Otherwise, the variables in the two arm schemas are unified as happens normally in Z when schemas are combined. The decoration, $Arm1$, that occurs in the instantiation, only applies to the schemas in the library. [Hayes and Wildman, 1993] choose this approach "because it separates the concerns of qualification of names and decoration of schemas". The framing schema for arm1 is simplified to

$$Arm1\_Ops \mathrel{\widehat{=}} \Delta Robot \wedge \Xi Robot\_0 \wedge arm2\text{-}Arm2 :: \Xi State$$

which allows changes to arm1's state variables only. Loading arm1 becomes

$$Load\_Arm1\_0 \mathrel{\widehat{=}} Generate\_Orientation \gg (arm1\text{-}Arm1 :: Load \wedge Arm1\_Ops)$$

3.5 Object-Z

Specifying an arm [see Fig. 3] in Object-Z is similar to the library specification. The specification is simpler because promoting operations in Object-Z is easier. This means that we do not bother to parameterise the arm class. The robot operations differ from those in the previous styles because each Object-Z operation schema has a delta (Δ) list that defines those state variables that potentially change. This simplifies the operations by removing the equating of variables that do not change. For example, *Extend* becomes a simple two line schema. Operations without a delta list inspect the state without causing any change.

$Component\_Loaded ::= loaded \mid unloaded$
$Arm\_Extent ::= retracted \mid extended$

Arm

$extent : Arm\_Extent$
$store : Component\_Loaded$

INIT
$extent = retracted$
$store = unloaded$

Retracted
$extent = retracted$

Extend
$\Delta(extent)$

$extent' = extended$

Retract
$\Delta(extent)$

$extent' = retracted$

Load
$\Delta(store)$

$extent = extended$
$store = unloaded$
$store' = loaded$

Unload
$\Delta(store)$

$extent = extended$
$store = loaded$
$store' = unloaded$

Figure 3: The arm class

$Robot\_Orientation ::= load\_arm1 \mid unload\_arm1 \mid load\_arm2 \mid unload\_arm2$

RobotBase

$orientation : Robot\_Orientation$

INIT: $orientation = load\_arm1$

Rotate

$\Delta(orientation)$
$new\_pos? : Robot\_Orientation$

$orientation' = new\_pos?$

$Arm1Orientation \mathrel{\widehat=} [orientation \in \{load\_arm1, unload\_arm1\}]$
$Arm2Orientation \mathrel{\widehat=} [orientation \in \{load\_arm2, unload\_arm2\}]$
$LoadArm1Orientation \mathrel{\widehat=} [orientation = load\_arm1]$
$LoadArm2Orientation \mathrel{\widehat=} [orientation = load\_arm2]$
$UnloadArm1Orientation \mathrel{\widehat=} [orientation = unload\_arm1]$
$UnloadArm2Orientation \mathrel{\widehat=} [orientation = unload\_arm2]$

Figure 4: The robotbase class

Robot

$base : RobotBase$
$arm1, arm2 : Arm$

$arm1 \neq arm2$
$arm1.extent = extended \Leftrightarrow base.Arm1Orientation$
$arm2.extent = extended \Leftrightarrow base.Arm2Orientation$

$Init \mathrel{\widehat=} arm1.Init \wedge arm2.Init \wedge base.Init$
$Rotate \mathrel{\widehat=} arm1.Retracted \wedge arm2.Retracted \wedge base.Rotate$
$Extend\_Arm1 \mathrel{\widehat=} arm1.Extend \wedge base.Arm1Orientation$
$Extend\_Arm2 \mathrel{\widehat=} arm2.Extend \wedge base.Arm2Orientation$
$Retract\_Arm1 \mathrel{\widehat=} arm1.Retract$
$Retract\_Arm2 \mathrel{\widehat=} arm2.Retract$
$Load\_Arm1 \mathrel{\widehat=} arm1.Load \wedge base.LoadArm1Orientation$
$Load\_Arm2 \mathrel{\widehat=} arm2.Load \wedge base.LoadArm2Orientation$
$Unload\_Arm1 \mathrel{\widehat=} arm1.Unload \wedge base.UnloadArm1Orientation$
$Unload\_Arm2 \mathrel{\widehat=} arm2.Unload \wedge base.UnloadArm2Orientation$

Figure 5: The robot class

The major difference occurs where operations are promoted to the robot level [see Fig. 4 and 5]. An intermediate level of structure is introduced via the *RobotBase* class that encapsulates all information about the robot's orientation. It serves to simplify the *Robot* class specification. No framing schemas are necessary and the load operation for arm1 is

$$Load\_Arm1 \;\widehat{=}\; arm1.Load \wedge base.LoadArm1Orientation$$

Each of the specification techniques achieves the same end via different methods. The following section discusses the positive and negative aspects of each technique.

4 Top-down Z specification styles

The preceding specifications of the production cell have all modelled the system as a collection of components with operations specified on a per component basis. However, the system can be viewed from the perspective of the metal blank with the production cell considered as a series of transitions between states. The transitions are equivalent to moving a metal blank from one component of the cell to another component, e.g., from the feed belt to the elevating rotary table. As well as transferring metal blanks from component to component, the movement of the robot, press, crane and elevating rotary table must be considered. Specifying these transitions leads to a specification that differs significantly from those earlier. Stylistically it is closest to the flat specification [Section 3.1]. The important difference is not the style used, but the model chosen, particularly as the model chosen changes the abstraction and granularity of the specification. The complete specification is presented in [MacDonald, 1998] and consists of given types, a state schema, an initialisation schema and twelve operation schemas. The twelve operation schemas are further subdivided into eight metal blank transition operations and four component movement operations.

The specification makes a further assumption to those outlined in [Section 2]. The assumption is that the extension and retraction of both the robot arms and the crane are hidden within the transitions. This extra assumption is because we choose not to model the robot's arms, just as in [Section 2] we choose not to model the crane's arm. For instance, an operation taking a metal blank from the elevating rotary table to the robot would have to deal internally with extending the robot's arm, turning on the magnet to pick up the metal blank and retracting the robot's arm.

4.1 State specification

The production cell's state can be divided into two separate groups. The first group contains a state variable for each component to record whether the component is loaded with a metal blank. Each of these is initialised to *unloaded*. The second group contains state variables related to component movement. These variables are present only for the robot, press, crane and elevating rotary table as these components do not have their movement hidden. The robot and elevating rotary table have their initial positions/orientations set in agreement with the original system requirements.

$$
\begin{array}{|l}
\hline
\textit{Cell} \\
\hline
\textit{feed\_belt\_store} : \textit{Component\_Loaded} \\
\textit{table\_store} : \textit{Component\_Loaded} \\
\textit{robot\_arm1\_store} : \textit{Component\_Loaded} \\
\textit{robot\_arm2\_store} : \textit{Component\_Loaded} \\
\textit{press\_store} : \textit{Component\_Loaded} \\
\textit{deposit\_belt\_store} : \textit{Component\_Loaded} \\
\textit{crane\_store} : \textit{Component\_Loaded} \\
\textit{table\_position} : \textit{Table\_Position} \\
\textit{press\_position} : \textit{Press\_Position} \\
\textit{robot\_orientation} : \textit{Robot\_Orientation} \\
\textit{crane\_position} : \textit{Crane\_Position} \\
\hline
\end{array}
$$

The state schema highlights one of the major differences of this specification, the high level of abstraction. For example, the robot is viewed as a simple entity that has two load attributes, *robot_arm1_store* and *robot_arm2_store*, and a position attribute, *robot_orientation*. The simple high-level view of the robot, as presented, is an abstraction from the complexity of the robot and its arms.

4.2 Operation specifications

An operation is specified for each transition of a metal blank from one physical component in the production cell to another. Operations are also specified for movement of the physical components. The operations presented below are only those that involve the robot. These operations are partial operations, and error schemas and total operations can be defined as mentioned earlier. The specification of the transitions from the elevating rotary table to the robot, robot to the press, press to the robot and robot to the deposit belt are introduced. Finally the operation to move the robot is presented.

Moving a metal blank from the elevating rotary table to the robot requires the robot to be oriented to *load_arm1*, the elevating rotary table to be in position to unload and the elevating rotary table loaded and the robot's arm1 unloaded. Post transition, the components have not moved but the robot's arm1 is loaded and the elevating rotary table is unloaded. No other state variables are changed.

$$
\begin{array}{|l}
\hline
\textit{ERT\_to\_Robot} \\
\hline
\Delta \textit{Cell} \\
\hline
\textit{table\_position} = \textit{ready\_to\_unload} \\
\textit{table\_store} = \textit{loaded} \\
\textit{robot\_orientation} = \textit{load\_arm1} \\
\textit{robot\_arm1\_store} = \textit{unloaded} \\
\textit{table\_store}' = \textit{unloaded} \\
\textit{robot\_arm1\_store}' = \textit{loaded} \\
\textit{feed\_belt\_store}' = \textit{feed\_belt\_store} \\
\textit{robot\_arm2\_store}' = \textit{robot\_arm2\_store} \\
\textit{press\_store}' = \textit{press\_store} \\
\textit{deposit\_belt\_store}' = \textit{deposit\_belt\_store} \\
\textit{crane\_store}' = \textit{crane\_store} \\
\textit{table\_position}' = \textit{table\_position} \\
\textit{press\_position}' = \textit{press\_position} \\
\textit{robot\_orientation}' = \textit{robot\_orientation} \\
\textit{crane\_position}' = \textit{crane\_position} \\
\hline
\end{array}
$$

With the robot loaded at orientation *unload_arm*1 and the press unloaded at position *open_for_arm*1, a transition can take place between the press and the robot. A metal blank is passed from the robot to the press changing only the load variables of the robot's arm1 and the press.

$$
\begin{array}{|l}
\textit{Robot\_to\_Press} \\
\hline
\Delta \textit{Cell} \\
\hline
\textit{robot\_orientation} = \textit{unload\_arm}1 \\
\textit{robot\_arm}1\textit{\_store} = \textit{loaded} \\
\textit{press\_position} = \textit{open\_for\_arm}1 \\
\textit{press\_store} = \textit{unloaded} \\
\textit{robot\_arm}1\textit{\_store}' = \textit{unloaded} \\
\textit{press\_store}' = \textit{loaded} \\
\textit{feed\_belt\_store}' = \textit{feed\_belt\_store} \\
\textit{table\_store}' = \textit{table\_store} \\
\textit{robot\_arm}2\textit{\_store}' = \textit{robot\_arm}2\textit{\_store} \\
\textit{deposit\_belt\_store}' = \textit{deposit\_belt\_store} \\
\textit{crane\_store}' = \textit{crane\_store} \\
\textit{table\_position}' = \textit{table\_position} \\
\textit{press\_position}' = \textit{press\_position} \\
\textit{robot\_orientation}' = \textit{robot\_orientation} \\
\textit{crane\_position}' = \textit{crane\_position} \\
\hline
\end{array}
$$

The operations *Press_to_Robot* and *Robot_to_DepositBelt* are similar in content and style to the preceding operations and are omitted for brevity.

The move operation, *Rotate_Robot*, receives as input the new position/orientation. The variable associated with the robot's movement, *robot_orientation*, is set to the new value and all other state variables are unchanged. The move operations for the other components are similar.

$$
\begin{array}{|l}
\textit{Rotate\_Robot} \\
\hline
\Delta \textit{Cell} \\
\textit{new\_pos}? : \textit{Robot\_Orientation} \\
\hline
\textit{robot\_orientation}' = \textit{new\_pos}? \\
\textit{feed\_belt\_store}' = \textit{feed\_belt\_store} \\
\textit{table\_store}' = \textit{table\_store} \\
\textit{robot\_arm}1\textit{\_store}' = \textit{robot\_arm}1\textit{\_store} \\
\textit{robot\_arm}2\textit{\_store}' = \textit{robot\_arm}2\textit{\_store} \\
\textit{press\_store}' = \textit{press\_store} \\
\textit{deposit\_belt\_store}' = \textit{deposit\_belt\_store} \\
\textit{crane\_store}' = \textit{crane\_store} \\
\textit{table\_position}' = \textit{table\_position} \\
\textit{press\_position}' = \textit{press\_position} \\
\textit{crane\_position}' = \textit{crane\_position} \\
\hline
\end{array}
$$

4.3 Summary

This specification differs significantly from those earlier introduced. Stylistically it is closest to the flat specification. The important difference is not the style used, but the model chosen. Particularly as the model chosen changes both the abstraction and granularity of the specification. The system is viewed from the perspective of the metal blank with the production cell considered as a series of transitions between states. This transition-based specification of the production cell abstracts away from detail to the extent that certain safety and ordering conditions become hidden. For example, the robot's arms are not present in the specification and therefore explicitly constraining the robot's arms to be retracted when rotating is not possible.

Each of the specification techniques achieves the same end via different methods. The following section discusses the positive and negative aspects of each technique.

5 Discussion

Initially, the discussion of each technique is constrained to its suitability for specifying the robot and its arms [Section 5.1]. However, the robot and its arms have a simple relationship and the components operate independently of each other at the robot level. [Section 5.2] deals with each technique when applied to the complete production cell with many components that interact to give the top-level system operations.

5.1 Structuring in the small

The size[6] of the robot specification differs little between the bottom-up techniques. The Object-Z version is the shortest at a page and a half while the parameterised version is the longest at two pages. An interesting point is that factoring components does not necessarily decrease specification size. In the top-down specification, it is not easy to separate the robot specification from the rest of the specification, but it is approximately two and a half pages long.

It can be noted that the partitioned style, which specifies each arm before inclusion in the robot state, does not work successfully for components that are dependent on some aspect of the higher-level state. This problem is avoided in the other component-based[7] styles by using generic parameters. A component-based specification requires the components to be independently understandable. The top-down specification by definition does not suffer from this problem.

The parameterised specification [Section 3.3] is the most difficult to understand of all the specifications with both framing schemas and hiding making the specification complex, while the Object-Z specification is probably the simplest to understand (once the Object-Z conventions are understood). The top-down specification is the most verbose, but whether this affects understandability is open to debate.

Looking at specific details of the specifications, there are no interesting differences between the specifications of the arms other than the simplification and lack of duplication gained by reusing the arm rather than specifying two similar arms. However, there are three kinds of difference at the robot level: differences relating to the robot's state schema and initialisation, differences relating to the *Rotate_Robot* operation, and differences in the arm operations at the robot level. The differences in the arm operations at the robot level centre on the different promotion techniques used by the structuring styles. The top-down specification will be considered and mentioned where appropriate in this comparison, but in some cases is too different for any useful detailed comparison to be made. In particular, the top-down specification does not explicitly specify the robot's arms and hence no comparison can be made.

[6] This is the size of the complete specifications, not just the fragments included in this article.

[7] A component-based specification style is a style that builds a larger specification from several smaller parts.

The main difference between the state schemas is the addition of predicates in the parameterised, library and Object-Z versions. These predicates make the constraints on the robot state explicit rather than implicit via the operations as illustrated in the simpler flat and partitioned versions. The initialisations are almost identical except for the instantiated generic arms in the parameterised style that require a framing schema to nominate which arm is being initialised. It can be inferred that parameterisation is not suited to structuring where multiple instantiations of the same type are needed, but is more suited where each instance is based on a different type.

Across the six different specification techniques, there are three distinct specification strategies used for *Rotate_Robot_0*. The first strategy is used in both the flat and parameterised versions and requires the post-state of every variable of the robot to be described explicitly. This example is from the parameterised version.

$$
\begin{array}{l}
\textit{Rotate\_Robot\_0} \\
\hline
\Delta \textit{Robot} \\
\textit{new\_pos}? : \textit{Robot\_Orientation} \\
\hline
\textit{arm}1.\textit{extent} = \textit{retracted} \wedge \textit{arm}2.\textit{extent} = \textit{retracted} \\
\textit{robot\_orientation}' = \textit{new\_pos}? \\
\textit{arm}1'.\textit{store} = \textit{arm}1.\textit{store} \wedge \textit{arm}2'.\textit{store} = \textit{arm}2.\textit{store} \\
\textit{arm}1'.\textit{extent} = \textit{arm}1.\textit{extent} \wedge \textit{arm}2'.\textit{extent} = \textit{arm}2.\textit{extent} \\
\hline
\end{array}
$$

The top-down specification is an extreme version of this first strategy where the post-state of each variable in the specification must be described explicitly. By hiding the arms the specification also weakens the pre-condition on a robot rotation as shown below.

$$
\begin{array}{l}
\textit{Rotate\_Robot} \\
\hline
\Delta \textit{Cell} \\
\textit{new\_pos}? : \textit{Robot\_Orientation} \\
\hline
\textit{robot\_orientation}' = \textit{new\_pos}? \\
\textit{feed\_belt\_store}' = \textit{feed\_belt\_store} \\
\textit{table\_store}' = \textit{table\_store} \\
\textit{robot\_arm}1\textit{\_store}' = \textit{robot\_arm}1\textit{\_store} \\
\textit{robot\_arm}2\textit{\_store}' = \textit{robot\_arm}2\textit{\_store} \\
\textit{press\_store}' = \textit{press\_store} \\
\textit{deposit\_belt\_store}' = \textit{deposit\_belt\_store} \\
\textit{crane\_store}' = \textit{crane\_store} \\
\textit{table\_position}' = \textit{table\_position} \\
\textit{press\_position}' = \textit{press\_position} \\
\textit{crane\_position}' = \textit{crane\_position} \\
\hline
\end{array}
$$

The second strategy is used in both the partitioned and library versions and uses the structure of the specification to equate whole components rather than individual state variables. The example used below is from the library style.

$$
\begin{array}{l}
\textit{Rotate\_Robot\_0} \\
\hline
\Delta \textit{Robot} \\
\textit{new\_pos}? : \textit{Robot\_Orientation} \\
\hline
\textit{arm}1\text{-}\textit{extent} = \textit{retracted} \wedge \textit{arm}2\text{-}\textit{extent} = \textit{retracted} \\
\textit{robot\_orientation}' = \textit{new\_pos}? \\
\theta \textit{arm}1\text{-}\textit{Arm}1 :: \textit{State}' = \theta \textit{arm}1\text{-}\textit{Arm}1 :: \textit{State} \\
\theta \textit{arm}2\text{-}\textit{Arm}2 :: \textit{State}' = \theta \textit{arm}2\text{-}\textit{Arm}2 :: \textit{State} \\
\hline
\end{array}
$$

The third strategy is used only in the Object-Z version. This version clearly identifies the components involved in the operation. Expanding that definition gives

$$
\begin{array}{|l}
\textit{Rotate} \\
\hline
\Delta(\textit{orientation}) \\
\textit{new\_pos}? : \textit{Robot\_Orientation} \\
\hline
\textit{arm}1.\textit{extent} = \textit{retracted} \wedge \textit{arm}2.\textit{extent} = \textit{retracted} \\
\textit{base}.\textit{orientation}' = \textit{new\_pos}? \\
\hline
\end{array}
$$

The Object-Z delta list enumerates the state variables that may be changed by this operation, removing the need to specify explicitly that other state variables do not change. The last two strategies for the rotate operation are more focussed as they abstract from unnecessary detail.

The arm operations at the robot level differ for each technique. The flat style, while being relatively easy to follow, has considerable duplication, no information hiding and does not encourage reuse. The partitioning style provides information hiding, but has the duplication problems associated with the flat technique. As already stated, parameterised structuring is not suited to multiple instantiations. Framing schemas are needed to equate the unused state components and to restrict the operation to the relevant arm. The parameterised style also uses hiding. While neither framing schemas nor hiding are difficult concepts, their use is counter-intuitive, especially when compared to the final two techniques. The library mechanism is considerably easier to understand with a framing schema needed only to equate the unused state components. Object-Z is simpler again with no need for framing schemas. The top-down specification does not have arm operations.

For a simple system where the components do not interact with each other (i.e. an arm1 operation at the robot level does not affect arm2 at all), the library mechanism or Object-Z seem to be the easiest to use and understand. The decision between them would be based on whether the specifier wishes to use an object-oriented approach or not. The top-down specification provides another perspective of the system which allows the specifier to abstract from the detail of the interaction between system components. This abstraction makes the top-down specification useful for initial understanding of the problem, but limited for completely understanding the complexities of interaction between system components.

5.2 Structuring in the large

The second half of the production cell specification involves combining all the components and promoting and combining component operations to operations on the production cell. Promotion is complicated by relationships between components at the production cell level. These relationships either impose additional constraints on operations defined for a single component or require defining system-level operations as combinations of component operations. Defining these relationships requires access to state variables of multiple components, but changing these variables is mostly not required.

The flat style is not appropriate as components are being considered. The arguments for not using the flat style at the robot level also apply at the production cell level. These include lack of information hiding and duplication that leads to a cumbersome specification (thirteen state variables with no more than five accessed by any single operation). For similar reasons, the top-down style, which is closely related to the flat style, is not considered in this section of the discussion. Components can be included in the production cell in two ways: as instances (declaration) or by direct inclusion (schema inclusion). The discussion of component inclusion is in two parts, the first considering alternatives in Z and the second considering Object-Z.

5.2.1 Inclusion in Z

Z schema inclusion promotes state variables but not operations. As was shown earlier using the parameterised style, such operations are normally promoted using framing schemas. For the production cell, only direct inclusion of components is discussed as the other methods of promotion (and/or combination) have been fully discussed in [Section 5.1] and either do not scale up (flat) or do not behave any differently in the large (partitioned and library). The state of the complete system, *Cell*, is a conjunction of the state from each of the individually specified components.

$$Cell \;\widehat{=}\; Feed\_Belt \land ERT \land Robot \land Press \land Deposit\_Belt \land Crane$$

Generating the system-level operations from the component operations using promotion and conjunction uncovers issues not present in a smaller, less complex specification. Firstly, extra constraints on operations at the production cell level are needed. For example a robot arm, as long as the robot is at a correct orientation, can always be extended within the robot component. However for safety reasons, the robot's arm extensions are constrained at the production cell level where the robot must interact with either the elevating rotary table, the press, or the deposit belt. Another issue is how to specify those state variables that are unchanged, especially when most of the production cell operations access state variables from only one or two components. The discussion focuses on the robot, but now considers its relationships with the rest of the cell.

5.2.1.1 Unchanged state variables

Consider the operation $Load\_Arm1\_1$, which is defined as the combination of the robot operation $Load\_Arm1\_0$ and the operation $Unload\_ERT\_0$ from the elevating rotary table. The operation $Load\_Arm1\_1$ changes the state variables associated with both the robot and elevating rotary table leaving the state variables associated with all other components unchanged. Equating unchanged components can be approached in several ways. The simplest method is to equate each of the unused states independently.

$$\begin{array}{|l}
\multicolumn{1}{l}{Load_Arm1_1} \\
\hline
\Delta Cell \\
Load_Arm1_0 \\
Unload_ERT_0 \\
\hline
\theta Feed_Belt' = \theta Feed_Belt \\
\theta Press' = \theta Press \\
\theta Deposit_Belt' = \theta Deposit_Belt \\
\theta Crane' = \theta Crane \\
\hline
\end{array}$$

However, with a large collection of components, this approach produces long lists and is a tedious process. It would be clearer if it were possible to equate the whole state except the components changed by the operation as shown below.

$$\begin{array}{|l}
\multicolumn{1}{l}{Load_Arm1_1} \\
\hline
\Delta Cell \\
Load_Arm1_0 \\
Unload_ERT_0 \\
\hline
\theta Cell' \setminus (Robot', ERT') = \theta Cell \setminus (Robot, ERT) \\
\hline
\end{array}$$

This is not valid Z as hiding is not allowed to be used with theta (θ). There are two equivalent methods that both involve declaring the sub-state before use. The first method uses existential quantification:

$$Cell\_0 \mathrel{\widehat=} \exists ERT \mathbin{⨟} Robot \bullet Cell$$

The second uses hiding and is closer to the preferred but invalid definition above. The definition for *Cell_0* should contain the state variables to be hidden in parentheses rather than the state schemas to which the variables belong. However, this minor extension captures clearly what is required.

$$Cell\_0 \mathrel{\widehat=} Cell \setminus (Robot, ERT)$$

Either method can be used in the following schema which is valid Z.

$$\begin{array}{|l}
\textit{Load_Arm1_1} \\
\hline
\Delta Cell \\
Load_Arm1_0 \\
Unload_ERT_0 \\
\hline
\theta Cell_0' = \theta Cell_0 \\
\hline
\end{array}$$

The production cell has eleven different sub-states for seventeen operations and having to pre-declare each sub-state and include a predicate in each operation is cumbersome. It is preferable to move the statement of what remains unchanged from the predicate section to the declaration section of a schema and avoid naming each substate. A new naming convention is proposed enabling the specifier to specify that the whole state remains unchanged except for a list of state variables. The list of state variables can also contain a schema reference where the schema reference is a state schema and it is equivalent to listing all the variables in the state schema. The new naming convention has the form $\Xi A\nabla(B)$ and is equivalent to $\Xi A \setminus (\Delta B) \wedge \Delta B$ or $(\exists \Delta B \bullet \Xi A) \wedge \Delta B$. The new naming convention is demonstrated by the definition of *Load_Arm1_1*.

$$Load\_Arm1\_1 \mathrel{\widehat=} \Xi Cell\nabla(Robot, ERT) \wedge Load\_Arm1\_0 \wedge Unload\_ERT\_0$$

where

$$\Xi Cell\nabla(Robot, ERT) \mathrel{\widehat=} \Xi Cell \setminus (\Delta Robot, \Delta ERT) \wedge \Delta Robot \wedge \Delta ERT$$

Using this new naming convention would have a large impact on the visible presentation, and arguably the understandability, of the top-down style. For example, the fourteen line specification of *Rotate_Robot* is reduced to a four line specification.

$$\begin{array}{|l}
\textit{Rotate_Robot} \\
\hline
\Xi Cell\nabla(robot_orientation) \\
new_pos? : Robot_Orientation \\
\hline
robot_orientation' = new_pos? \\
\hline
\end{array}$$

5.2.1.2 Extra constraints

To capture additional constraints on system-level operations, extra predicates are added to operations at the production cell level. Adding extra predicates is quite straightforward and an example is the schema *Extend_Arm1_1*. *Extend_Arm1_1* constrains the arm to extend only if a collision can not occur between the loaded arm and a blank on either the elevating rotary table or in the press.

$$
\begin{array}{|l}
\hline
\textit{Extend\_Arm1\_1} \\
\Xi Cell \nabla (Robot) \\
Extend\_Arm1\_0 \\
\hline
(robot\_orientation = load\_arm1 \\
\qquad\qquad \wedge\ table\_position = ready\_to\_unload \\
\qquad\qquad \wedge\ table\_store = loaded \Rightarrow arm1\_store = unloaded) \\
(robot\_orientation = unload\_arm1 \\
\qquad\qquad \wedge\ press\_position = open\_for\_arm1 \\
\qquad\qquad \wedge\ arm1\_store = loaded \Rightarrow press\_store = unloaded) \\
\hline
\end{array}
$$

5.2.1.3 Error schemas

The operations defined previously for the production cell are partial operations. An operation is partial when its behaviour is defined for only some states. Error schemas are specified and then combined with partial operations to produce total operations that are always defined and these either output an *ok* report or a report specifying the error that occurred. This means the behaviour of the operation is specified in all circumstances. The error schema and the total operation for loading arm1 are shown as an example.

$$Report ::= ok \mid wrong\_robot\_orientation \mid avoid\_collision\_between\_blanks \mid \cdots$$
$$Success \mathrel{\widehat=} [r! : Report \mid r! = ok]$$

$$
\begin{array}{|l}
\hline
\textit{Extend\_Arm1\_Error} \\
\Xi Cell \\
r! : Report \\
\hline
robot\_orientation \notin \{load\_arm1, unload\_arm1\} \Rightarrow \\
\qquad r! = wrong\_robot\_orientation \\
robot\_orientation = load\_arm1 \Rightarrow \\
\qquad (table\_position = ready\_to\_unload \wedge table\_store = loaded \\
\qquad \wedge\ arm1.store = loaded \Rightarrow r! = avoid\_collision\_between\_blanks) \\
robot\_orientation = unload\_arm1 \Rightarrow \cdots \\
\hline
\end{array}
$$

$$Extend\_Arm1 \mathrel{\widehat=} (Extend\_Arm1\_1 \wedge Success) \vee Extend\_Arm1\_Error$$

5.2.2 Object-Z

Object-Z can be structured using either inheritance or instantiation; for this case study, instantiation is used because it models the production cell more naturally as a collection of structured components.

5.2.2.1 Unused state variables

Promoting individual component operations to the production cell and combining them to form production cell operations is simple in Object-Z, as shown in the robot example, with none of the Z problems of having to equate unused state variables.

5.2.2.2 Extra constraints

Adding extra constraints to an operation is not as simple and can be attempted in several ways. The first adds an extra local operation containing the extra predicates. An example of an extra constraint and its use is shown for the extend arm operation that restricts the arm's extension to avoid collisions between blanks.

$$
\begin{array}{|l}
\textit{Extra\_Extend\_Arm}1 \\
\hline
(robot.base.orientation = load\_arm1 \land ert.position = ready\_to\_unload \\
\qquad \land\ robot.arm1.store = loaded \Rightarrow ert.store = unloaded) \\
(robot.base.orientation = unload\_arm1 \land press.position = open\_for\_arm1 \\
\qquad \land\ robot.arm1.store = loaded \Rightarrow press.store = unloaded) \\
\hline
\end{array}
$$

$$Extend\_Arm1 \mathrel{\widehat=} robot.Extend\_Arm1 \land Extra\_Extend\_Arm1$$

In some cases, the extra constraints could be operations exported from one of the component classes. This was demonstrated in the *Robot* class definition where the operation *Arm*1*Orientation* from the *RobotBase* constrains the *Extend_Arm*1 operation. This method has some drawbacks; the extra constraints may not be able to be placed in a single class. The constraint *Extra_Extend_Arm*1 is an example because of inter-object dependencies. A possible solution is for the component classes to provide an operation equivalent to each possible combination of state variables and every possible value of the state variable and combine these essentially boolean operations (predicates) at the top-level. For the arm class, the additional operations would include the following:

$$
\begin{array}{|l}
\textit{Arm\_Retracted} \\
\hline
extent = retracted \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{Arm\_Extended} \\
\hline
extent = extended \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{Arm\_Loaded} \\
\hline
store = loaded \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{Arm\_Unloaded} \\
\hline
store = unloaded \\
\hline
\end{array}
$$

Combining simple operations to capture the extra constraints has disadvantages. The combination of simple operations generates a schema of approximately the same size and arguably greater complexity than the extra operation. Placing the simple operations in the classes is conceptually sound, but increases the complexity of the class without an equivalent lessening of the complexity at the higher level. Placing an extra operation within a component class also causes problems by placing application-specific information within a low-level, reusable class. This can be overcome using inheritance and placing the extra operations in an intermediate class. The intermediate class is a wrapper around the reusable class customising it for the application.

The extra constraints could become class invariants. This may require rewriting the predicates (from the extra operations) to ensure the system is not over-constrained. The constraint *Extra_Extend_Arm*1 is suitable as part of a class invariant. Adding extra predicates to the class invariant makes them apply to all operations in the class. However, the constraints may be required only for a subset of the class operations and this can make the specification harder to understand. Some constraints only need hold for the duration of an operation and not at any other time. For example, moving the press requires the robot's arms be retracted and yet loading the press requires a robot arm to be extended. In our specification, it is not possible to capture as a class invariant that the robot arms must be retracted if the press is moving, yet are allowed to be extended at other times.

5.2.2.3 Error handling

Error handling in Object-Z has the same problems and solutions as extra constraints. Adding error schemas to the production cell highlights a stylistic problem. Unlike the familiar Z style that embeds schemas in informal text, the Object-Z class syntax does not encourage specifiers to include informal explanations within the class. Instead, class definitions are embedded in informal text. This means that each class needs to be kept as small as possible. This can be achieved by using inheritance and instantiation to build complex classes in stages.

6 Conclusions

This article has investigated six different structuring techniques for Z specifications. The first five techniques were introduced and compared using part of the production cell specification. The specification of the robot is small enough to understand and yet complex enough to enable comparison of the structuring techniques. The same structuring techniques were also applied to a larger context, the complete production cell, to investigate if the approaches that are appropriate in the small are also valid in the large. The introduction of the first five techniques focused on a component-based view of the production cell; the sixth technique is introduced using a transition-based view of the production cell. This last technique developed a specification that differs significantly from those developed using the the first five techniques.

Each structuring technique has benefits and is suited to different situations. The flat technique is suitable for small specifications where the use of components is unnecessary. Each component of the production cell except the robot was built in this fashion except in the Object-Z version. The partitioned technique provides a suitable style when each component is able to be independently specified. Structuring using parameterisation is useful if the instances have distinct parameters, but quickly becomes complex for multiple instances with the same parameter. The library structuring method embodies the advantages of the previous two techniques. The library mechanism overcomes the lack of reuse in the partitioned style and the instantiation problems of the parameterised style. With a larger specification, the focus moved from structuring to promotion difficulties and the promotion technique used by the partitioned and library techniques was found to be suitable. An extension to Z was suggested and involved the extension of the Xi (Ξ) naming convention. The new naming convention enables the specifier to specify that the whole state remains unchanged except for the list of state variables following the symbol, ∇. Object-Z provides a concise structuring and promotion method for small systems, but in larger systems some problems were encountered dealing with extra constraints on top-level operations and with error schemas. Stylistically the sixth specification is closest to the flat specification. The important difference is not the style used, but the model chosen. In particular, the new model changes both the abstraction and granularity of the specification. Our transition-based specification of the production cell abstracts away from detail to the extent that certain safety and ordering conditions become hidden.

The six specifications are specifications of the production cell's expected behaviour, not a specification of a possible controller implementation. Consequently, the specification does not capture the optimal order of robot rotation or force the press to actually press a metal blank. Furthermore, the scheduling of operations to control the production cell efficiently is not captured. The comparison of the specification styles has focused on their use rather than their comprehensibility to readers. A comparison of the comprehensibility of the specifications would substantive, but those interested should read [Finney, 1996] for a discussion of the comprehensibility of formal methods.

Specifiers and designers need to be aware of a range of structuring techniques and understand when each technique is applicable. This article has provided some realistic examples of techniques currently in use. As Z is applied to larger specification tasks, well-structured specifications become even more important.

References

[Duke and Rose, 2000] R. Duke and G. Rose. *Formal Object-Oriented Specification using Object-Z*. MacMillan, 2000.

[Finney, 1996] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[Hall, 1990] A. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*, pages 290–318. Springer-Verlag, 1990.

[Hayes and Wildman, 1993] I. Hayes and L. Wildman. Towards libraries for Z. In J. Bowen and J. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting, London, December 1992*, Workshops in Computing, pages 37–51. Springer-Verlag, 1993.

[Hayes, 1993] I. Hayes, editor. *Specification Case Studies*. Prentice Hall, London, UK, 2nd edition, 1993.

[Kernighan and Plauger, 1978] B. Kernighan and P. Plauger. *The Elements of Programming Style*. McGraw-Hill, second edition, 1978.

[Lano and Houghton, 1994] K. Lano and H. Houghton, editors. *Object-oriented specification case studies*. Prentice Hall, 1994.

[Lewerentz and Lindner, 1995] C. Lewerentz and T. Lindner. Case study production cell. In *Formal Development of Reactive Systems*, number 891 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[MacDonald and Carrington, 1994] A. MacDonald and D. Carrington. Z specification of the Production Cell. Technical Report TR94-46, Software Verification Research Centre, Dept. Computer Science, Univ. of Qld, Australia, November 1994.

[MacDonald and Carrington, 1995] A. MacDonald and D. Carrington. Structuring Z Specifications: Some Choices. In J. Bowen and M. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, number 967 in Lecture Notes in Computer Science, pages 203–223. Springer-Verlag, September 1995. Also available as SVRC Techreport, TR95-19.

[MacDonald, 1998] A. MacDonald. *Designing Software from Formal Specifications*. PhD thesis, Department of Computer Science and Electrical Engineering, University of Qld, Australia, 1998. Available from Australian Digital Theses Program: http://adt.caul.edu.au/.

[Rose, 1992] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[Schuman and Pitt, 1987] S. Schuman and D. Pitt. Object-oriented subsystem specification. In L. Meertens, editor, *Program Specification and Transformation*, pages 313–341. North-Holland, 1987.

[Smith, 1999] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.

[Spivey, 1989] J. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.

[Spivey, 1992] J. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[Stepney *et al.*, 1992] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

[Toyn, 1999] I. Toyn, editor. *Z Notation: Final Committee Draft, CD 13568.2*. August 1999.